A Transformation to Convert Packing Code to Compact Datatypes for Efficient Zero-Copy Data Transfer

Fredrik Kjolstad Torsten Hoefler Marc Snir
University of Illinois at Urbana-Champaign
{kjolsta1,htor,snir}@ilinois.edu

Abstract

Many high performance applications spend considerable time packing data into contiguous communication buffers. Datatypes provide an alternative by describing the layout of the communicated data. This empowers the runtime system to retrieve non-contiguous elements directly from application data structures. However, programmers find complex datatypes hard to use and are reluctant to invest time and effort to rewrite packing code to datatype code. Fortunately, the transformation from packing code to datatypes can be automated, and the programmer can replace packing code with datatypes at the push of a button. The transformation allows easy porting of applications to new machines that benefit from datatypes, thus improving programmer productivity.

We present an algorithm for converting packing code to datatype code based on a novel IR and a suite of optimizations. We have implemented the algorithm in a tool that transforms C packing code to an MPI datatype, and rewrites the packing code consumer to instead use the datatype. Our evaluation shows that our algorithm is applicable to real-world packing code, that it is fast enough to be used interactively, and that the datatypes it produces are compact and well optimized. Finally, we evaluate the performance of the code produced by our tool, showing that it outperforms the original packing code on a state-of-the art system. However, the real benefit of datatypes is in the future, when hardware support for non-contiguous gather-scatter transfers becomes ubiquitous.

1. Introduction

Data movement is a fundamental part of parallel applications on distributed memory machines. It is also often the most costly operation in terms of time and energy, and usually does not scale as well as the rest of the application. It is therefore essential that we move data as few times as possible, and move it efficiently when data movement cannot be avoided.

Datatypes are objects that describe data layout. Examples include strided layouts and indexed layouts. In strided layouts each block is located a fixed stride from the previous block. In indexed layouts each block's location is described by a displacement from the first block. When datatypes are used with a data movement operation such as send, receive, put or file write, non-contiguous data can be transferred with a single call to the underlying runtime.

University of Illinois at Urbana Champaign Technical Report 2011.

Datatypes are powerful tools that enable runtimes to optimize data movement in several ways. First, multiple blocks can be packed into a contiguous buffer and sent in one operation. Even with the additional copy pass, this is often a significant improvement over multiple small blocking transfers due to the high latency cost associated with each transfer. Such packing can also be performed by user packing code. This is a common idiom in scientific applications [10]. However, by describing non-contiguous data using datatypes, the runtime is free to determine the best way to transfer the data for a given system. It can optimize data packing [4, 9, 19], or use an alternative communication scheme, such as pipelining small asynchronous messages [7].

Moreover, many network controllers support direct memory access (DMA) and can access user memory without involving the CPU. Some networks, such as InfiniBand [11], also provide support for non-contiguous data transfers. This is a very powerful feature that enables zero-copy transfers [20, 26], which avoids memory-to-memory data copying. This can significantly reduce communication overheads [10, 20], but is only possible if the data layout is provided as an argument in the communication call.

Finally, since datatypes allow known data layouts to be specified declarative and concisely they increase the readability of the code. This is important to reduce the cost of software maintenance, an activity that may span decades.

MPI is a parallel programming model where a rich set of datatypes are available to the programmer. The programmer can specify contiguous and vector (strided) datatypes, as well as homogeneous and heterogeneous indexed datatypes. Furthermore, the programmer can describe complex data layouts efficiently by composing datatypes hierarchically.

A more restricted set of datatypes is available in several network API's, such as BSD Sockets, ARMCI [16] and Infiniband OFED [18]. They are also planned for inclusion in GASNet 2.0 [2], which is the target API for the Berkeley UPC Compiler.

However, datatypes are often underused [10, 19]. We believe there are at least two reasons for this. First, runtimes were traditionally not optimized for datatypes and datatypes were therefore often slower than manual packing loops. However, this has changed and runtimes are now better optimized [4, 9, 19], and provide powerful features such as non-contiguous zero-copy transfers that can only be exploited if datatypes are used [20, 26]. Second, datatypes are considered hard to construct. They are declarative, and require programmers to precisely describe exact data layouts in a terse way. Furthermore, they are very hard to debug since no tools are available to track the memory locations accessed by a communication library. However, once constructed their terseness and preciseness makes them easy to reason about. In short, datatypes declare the structure of the data instead of the logic required to pack it.

Moreover, parallel programming models such as UPC [24] and CAF [17] do not expose datatypes to the programmer (although

a UPC extension exist that implicitly supports datatypes through indexed and strided copies). Therefore, the programmer has no other choice than to express non-contiguous data transfers using either multiple sends or packing code.

The potential performance, performance portability and readability benefits of using datatypes, combined with the difficulty in creating them for humans and the lack of means to express them in some languages, motivates an automated approach. Since packing code is a common idiom for sending non-contiguous data, a technique for converting packing code to datatype code is needed.

For programming models that expose datatypes to the programmer, a refactoring tool or a source-to-source compiler should be used to port packing code to datatype code. For programming models where datatypes are not exposed to the programmer, such as UPC and CAF, a compiler pass that converts packing code to datatype code is desirable. This allows the compiler to output code that takes advantage of datatype capabilities in the network or uses pipelined asynchronous messages where this is supported, or that optimizes the packing for the target architecture where it is not.

We present a novel algorithm that converts packing code to datatype code. The algorithm converts the packing code to an intermediate representation (IR) called the Datatype IR. The IR compactly captures the information required to generate datatypes. The algorithm then performs a number of specialization and compression passes to optimize the IR so that compact datatypes can be emitted. After the algorithm has produced a datatype description of the layout of the packed data, the IR can be used to replace the existing packing code with datatype code. The presentation assumes C and MPI, but the techniques generalize to other environments.

We implemented our algorithm as a refactoring plugin for C with MPI on top of Eclipse CDT. We used this implementation to evaluate the approach on the NAS Parallel LU Benchmark. The evaluation shows that the algorithm is applicable to real world code and that it finds good datatypes.

2. Related Work

Gojun et al., developed a pre-processor tool called AutoMap that automatically generates datatypes for user-annotated C structs [8]. Moreover, Tansey & Tilevich developed a GUI tool that can generate datatypes for C++ classes [23]. These tools automate the generation of datatypes for struct and class definitions, but do not look for opportunities to use these datatypes in client code or for structured accesses to arrays that can be replaced with vector or contiguous types. In contrast, our technique generates datatypes based on access patterns in packing code, finds indexed and vector accesses in arrays, and rewrite the client code to use these datatypes.

There is a large body of research on optimizing the performance of datatypes. Gropp, Lusk and Swider provide a taxonomy of MPI datatypes according to their memory reference patterns, and demonstrate how to efficiently implement these patterns using a variety of techniques [9].

One line of research on datatype processing aims to improve the performance of datatype packing in MPI implementations over user packing code, by using efficient internal data structures, runtime and machine information. Bynna et al. present a technique to improve the performance of derived datatypes, by automatically chosing a packing algorithm that is optimized for the memory-access cost of the target machine [4]. Ross, Miller and Gropp describe an efficient internal representation of datatypes called dataloops that aids MPI implementation that performs datatype packing in maintaining high performance during datatype processing [19].

A second line of research describes techniques to take advantage of datatypes to exploit advanced network features that allows moving non-contiguous data without any packing. Wu, Wyckoff and Panda compare the performance of an MPI implementation that performs datatype packing and an implementation that uses the Remote Direct Memory Access (RDMA) feature of InfiniBand [11] to avoid either the packing or the unpacking involved in transmitting non-contiguous [26]. Santhanaraman, Wu and Panda presents a technique they call Send Gather Receive Scatter (SGRS) that uses InfiniBand channels to avoid both the packing and unpacking involved in sending/receiving non-contiguous data [20]. Finally, Tanabe and Nakajo developed an hardware accelerator for datatypes, called DIMMnet-2, that can transfer non-contiguous data.

Researchers	Approach	Speedup
Wu, et al. [26]	Infiniband non-contiguous remote load/store	3.6x
Santhanaraman, et al. [20]	Infiniband non-contiguous channel communication	4.8x
Worringen, et al. [25]	SCI non-contiguous copy to global memory	2.1x
Tanabe and Nakajo [22]	DIMMnet-2 support for non-contiguous RDMA	6.8x

Table 1. Previous work on speeding up datatypes through hardware support, with maximum speedups reported by researchers.

All of these techniques require datatype to be specified explicitly. Our technique converts packing code to datatype code, thereby enabling these optimizations.

Previous work has established that message bandwidth can be increased significantly if hardware support for zero-copy transfer of non-contiguous data is provided. Taking advantage of such features requires datatypes to be specified. Wu, et al. showed vector bandwidth improvements up to a factor of 3.6 for large messages compared to manual packing code when using their Multi-W technique for InfiniBand [26]. Similarly, Santhanaraman, et al. showed vector bandwidth improvements up to a factor of 4.75 for large messages using their SGRS technique for InfiniBand [20]. They also report low CPU utilization as communication workload is offloaded to the network, which indicates increased potential for exploiting communication-computation overlap. Furthermore, Worringen, et al. report performance improvements up to a factor of 2.1, from taking advantage of hardware support in the SCI interconnect to copy non-contiguous data directly to global memory. Finally, Tanabe and Nakajo demonstrated a performance improvement up to a factor of 6.8 from accelerating MPI Datatypes using their DIMMnet-2 RDMA system [22]. Table 1, summarizes previous work on speeding up datatypes through hardware support.

Furthermore, multiple research groups have demonstrated that datatypes can have comparable or better performance than manual packing code, even when the network does not support non-contiguous transfers. Ross et al. showed that an optimized MPI implementation can have comparable performance to manual packing code when transmitting common data structures such as vectors and 3D faces [19]. Moreover, Byna, et al. provide a technique that outperforms manual packing code by as much as 205% for a matrix transpose by taking advantage of knowledge of the memory system to improve memory access cost [4]. Hoefler and Gottlieb demonstrate speedups up to a factor of 3.8 and 18% for a Fast Fourier Transform and a conjugate gradient solver respectively by expressing communication using datatypes [10].

This firmly establishes datatypes as a useful way of specifying non-contiguous transfers that can provide speedups given the right hardware and/or software. Furthermore, we expect network capabilities to improve in the future making datatypes increasingly relevant.

Key steps in our algorithm is the specialization from indexed types to vector types, and from vector types to contiguous types. These steps rely heavily on algebraic expression simplification and

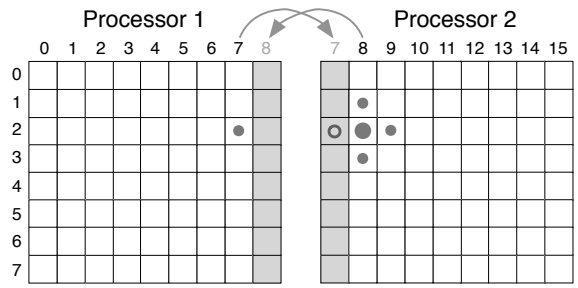


Figure 1. 5-point stencil Computation on two processors with ghost cells (grey) and border exchanges.

loop induction variable summarization. Both of these problems have been extensively studied in the literature.

Algebraic expression simplification is widely used in mathematical packages such as Matlab and Mathematica. Moses presents various techniques to reduce the size of expressions with the dual goals of making them more intelligible to the user and to allow the designer to construct useful and efficient systems [14]. Furthermore, Buchberger and Loos formally describe the problem of canonical algebraic simplification and then present two major groups of simplification techniques [3]. Our implementation of the datatype extraction algorithm uses the Open Source Symja library [21] to simplify expressions.

Loop induction variable detection and summarization has most commonly been studied in the context of operator strength reduction. Induction variables are variables whose values change between iterations of a loop. Linear induction variables are induction variables whose values follow an arithmetic sequence. That is, they increase or decrease by a fixed amount from one loop iteration to the next. Cocke and Kennedy developed an early algorithm for operator strength reduction based on induction variables [5]. Cooper et al. later improved on this algorithm by taking advantage of SSA form to perform efficient sparse induction variable detection [6]. Our implementation of loop induction variable summarization is based on the technique described by Muchnic [15].

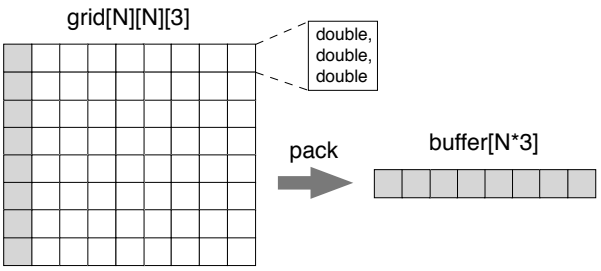
3. Illustrative Example

We present a running example to illustrate some of the challenges of converting packing code to datatype code. The running example is a border-exchange between two nodes that execute a two dimensional iterative stencil computation. The example is similar to one of the simpler packing loops in the NAS LU benchmark. The example is implemented in C99 using MPI. As mentioned before, MPI provides a rich API for constructing datatypes and is therefore an excellent target for our algorithm. We first provide a quick introduction to MPI datatypes in section 3.1, before we describe the details of the border exchange in section 3.2.

3.1 MPI Datatypes

MPI Derived Datatypes are opaque objects that describe data layout. They define a type map, which is an ordered sequence of primitive datatypes and displacements. Primitive datatypes correspond to the primitives in C and Fortran, and include such types as `MPI_DOUBLE` and `MPI_INT`. Datatypes are a central concept in MPI and are used with message sends or receives, remote direct memory access (one sided), and file IO to specify the data to read or write.

MPI provides several constructors that can be used to construct derived datatypes. These include `MPI_Type_create_hindexed()` and `MPI_Type_vector()`. The former creates a derived datatype given a displacement list, while the latter creates a derived datatype that describes a vector with a given count, block length and stride.



(a) The left Ghost Cell Halo is packed into the send buffer. Each cell contains three values

```
1 double buffer[N * 3];
2 for(int i=0; i < N; i++) {
3   buffer[3*i]   = grid[i][0][0];
4   buffer[3*i+1] = grid[i][0][1];
5   buffer[3*i+2] = grid[i][0][2];
6 }
7 MPI_Send(buffer, N * 3, MPI_DOUBLE, left, tag, comm);
```

(b) C99 packing code that copies the left ghost cell halo into a send buffer

```
1 MPI_Datatype vec_t;
2 MPI_Type_vector(N, 3, N * 3, MPI_DOUBLE, &vec_t);
3 MPI_Type_commit(&vec_t);
4 MPI_Send(&grid[0][0][0], 1, vec_t, left, tag, comm);
5 MPI_Type_free(&vec_t);
```

(c) A vector that describes the layout of the data to be sent

Figure 2. Example where the left ghost cell halo is sent

3.2 Border Exchange Example

Iterative stencil computations on structured grids are a common class of algorithms. They are used in weather and atmospheric simulations, fluid dynamics, and many other applications. The input to many stencil computations is a regular, N-dimensional mesh. The values of the mesh points are updated iteratively based on the values of surrounding points from the previous iteration. All points use surrounding points in the same relative positions. This set of relative positions is called a *stencil*.

Such computations are implemented on distributed memory systems by partitioning the mesh, and adding *halo points* at the boundary of each partition. Halo points are replicas of border points from neighboring partitions that are needed to compute the local updates. The halo region is updated between every iteration in a process called a border exchange [12].

Figure 1 illustrates a stencil computation on two processors with a 5-point stencil. Each point contains three double values, and the computation uses border exchanges to communicate border values. The white cells form the domain of the computation. The gray area shows the ghost cell region of each processor.

Figure 2(a) illustrates how the left ghost cell region of processor 2 is packed into a communication buffer, and figure 2(b) shows the corresponding C99 source code. The code consists of a loop with index `i` that is used to address both the source and the destination of the packing operations. Inside the loop there are three packing statements that pack the three values of each border cell into the packing buffer.

Figure 2(c) shows the equivalent MPI datatype code. The source code constructs a vector with N blocks, 3 elements per block, and an $N*3$ element stride between each block. The vector is then used

to parameterize the `MPI_Send` and specifies the layout of the data to send, relative to location `&grid[0][0][0]`.

4. Datatype Generation

The algorithm described in this section automates the conversion from packing code to datatype code. That is, given packing the code on lines 2–6 in figure 2(b) it produces the datatype in figure 2(c). The input to this algorithm is a set of packing statements and packing loops, and the output is a datatype that describes the layout of the data that is packed. In a refactoring tool the programmer can provide the input packing statements, while in a compiler they can be identified through a reaching definitions analysis. This analysis would summarize statements that define array values that are subsequently sent, or that use array values that are received.

For ease of presentation, we only discuss packing code that copies data from user memory to a communication buffer. Generating datatypes for unpacking code is totally symmetric and is handled in the same way.

The problem of generating datatypes from packing statements can be broken down to a sequence of sub-problems that can be solved independently. Central to this process is an intermediate representation that we call the Datatype IR. The IR and is construction is discussed in section 4.1. Once the IR has been constructed a number of optimizations are applied to yield a compact datatype. An optimized datatype, as demonstrated by our evaluation, leads to significantly more efficient code than unoptimized datatypes. The datatype optimization transformations are described in section 4.2. Figure 4 shows three examples of the Datatype IR, in different stages of optimization.

4.1 Datatype IR

The Datatype IR is constructed from imperative languages with packing code, such as C, C++, Fortran and UPC. It captures the necessary information to generate compact datatypes, and greatly simplifies the optimizations. Once constructed, it describes a datatype that is equivalent to the packing code, and can be used to replace the packing code with a code that constructs a datatype. The Datatype IR is the focal point of our algorithm and serves to simplify each stage, and is specified in section 4.1.2.

However, before the Datatype IR can be constructed, a number of preconditions must be checked. The preconditions are described in section 4.1.1, and must be met before our algorithm can create datatypes and rewrite code. The algorithm can convert any packing code that satisfies these preconditions to datatype code. Finally, section 4.1.3 describes how the IR can be constructed from packing code written in C. Once the initial IR has been generated, it is optimized by subsequent passes (see section 4.2).

4.1.1 IR Preconditions

The algorithm targets packing code blocks: a sequence of statements that copy data from program data structures into a buffer that is consumed by the target send call. The algorithm checks that the following preconditions are met before it generates the IR:

1. The code block consist of: nested loops, assignments and conditionals statements
2. The code block writes into consecutive locations in the buffer

Note that in packing code blocks where the third condition does not hold it can often be established by an initial step where the packing code is reordered. The reordering involves known techniques such as code motion and loop transformations (loop split and loop reversal). These have been studied extensively in the literature, and are outside the scope of this work. Furthermore, in real-world code that we have studied precondition three holds in the vast majority of cases, and can easily be established with at most one loop split where it does not hold.

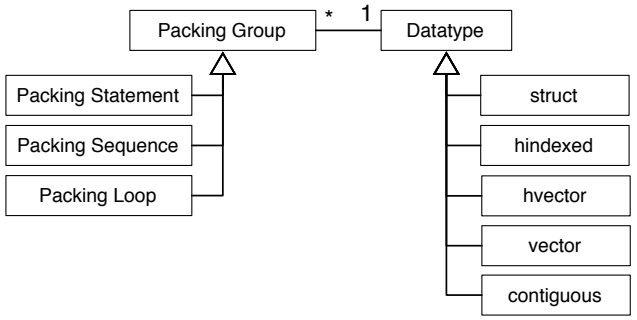


Figure 3. Datatype IR

4.1.2 Datatype IR Specification

The Datatype IR is defined recursively and describes the relationship between *packing groups* in the source code and datatypes. Figure 3 depicts the Datatype IR with the most important datatypes. However, as discussed in section 4.2.8 other datatypes are possible. A Packing Group represents packing code and is one of:

Packing Statement A statement that copies a value from user memory to a packing buffer. The IR node for a packing statement contains the memory location the value is retrieved from, as well as the location in the packing buffer it is packed into.

Packing Sequence A sequence of packing groups. The IR node for a packing sequence contains the sub-groups, as well as expressions that describe the symbolic *distance* between every pair of consecutive sub-groups in the source data structure. Section 4.1.3 describes how distances are calculated.

Packing Loop A loop whose body is a packing group. The IR node for a packing loop contains the body sub-group as well as a summary of loop induction variables and loop constants. Induction variables are discussed in section 4.1.3.

Every packing group has a *source location* associated with it. The location describes the first location in the source data structure that the packing group reads from. Thus, the location of a packing group is identical to the location of its first packing statement. The location of a packing statement is a pair of the form $(name, displacement)$. The first element is a named variable or pointer that is used to access the source data structure. The second element is an expression that describes the offset from the name to where the packed element is located. The displacement could either be a integer offset expression, in the case of arrays, or a field describing the struct element that is accessed. For example, the location of the following packing statement `b[j] = a[i+1]`, where `b` is a packing buffer, is the pair $(a, i + 1)$. This would also be the location of the equivalent packing statement `b[j] = *(a+i+1)`.

A more complex example is `b[i] = a[i][j][2]`, where the array a is defined as $a[D_1][D_2][D_3]$. In this case the location is determined by symbolically multiplying each array subscript by the size of every inner dimension, and then adding these products together. That is, given the source data structure access expression $A[s_n]...[s_1][s_0]$, the location expression is symbolically computed using the formula $\sum_{j=0}^{n} (s_j \prod_{k=0}^{j-1} dim_k)$. The location of the example thus becomes $(a, iD_2D_3 + jD_3 + 2)$.

In the IR each packing group is mapped to exactly one datatype, while a datatype can describe multiple packing groups. The datatypes in our IR match the datatypes in MPI. These form a powerful set of datatypes that can be composed to describe any data layout,

so the same IR can handle other environments. The supported datatypes are, ordered from more general to more efficient: struct, hindexed, hvector, vector and contiguous. Each datatype will be discussed further in section 4.2, as the optimizations that produce them are detailed.

4.1.3 Datatype IR Construction

The Datatype IR can be constructed from any imperative language with packing code, but we will discuss it for C with MPI. As described earlier, the input to the algorithm is a set of packing statements and packing loop. From these, packing sequences and packing loops are constructed. Every packing group inside a packing loop, or at the top level, is added to the same packing sequence. Additionally, packing groups inside loops are added to a new packing loop node that represents that loop.

For each packing sequence the distance between every consecutive pair of sub-groups is computed. The distance between two packing groups is only defined if their locations are in the same array, and none of the index expressions are conditionally redefined between the packing groups. If these conditions hold, then the distance is the location index expression of the second packing group minus the location index expression of the first group. For example, given two packing statements that pack the variables at location `a[i]` and `a[i+k]` the distance is $(i + k) - i = k$, assuming `i` is not redefined between the packing statements. If it is redefined then the index expression of the second packing statement must be updated to reflect this before the distance is computed. That is, if `i` was incremented by one between the packing statements, then the distance would be $(i + 1 + k) - i = k + 1$. In general this would require reaching definitions analysis. However, precondition 1 require the code to be "straight line code" (perfectly nested loops without conditionals) which makes the analysis straight forward.

For each packing loop, loop constants and loop induction variables are summarized using standard approaches [5, 15]. A discussion of these can be found in section 4.2.2.

Finally, to complete the IR the packing groups are assigned datatypes. Structs are the most general datatypes, and can be used to describe the layout of the data packed by any packing code that passes the preconditions in section 4.1.1. The IR construction stage therefore assigns a struct datatype to each of the packing groups. Figure 4(a) shows the resulting IR for our border exchange example. The loop is represented by a struct that conceptually contains N sub-structs, one for each iteration of the loop. However, only one struct node representing the N sub-structs is actually constructed in the IR. This sub-struct represents the packing sequence in the loop body and have 3 double sub-types, one for each packing statement.

4.2 Datatype Optimization

The previous section described how to construct an IR representing a correct datatype. Code can be emitted to construct a hierarchy of structs, and each packing statement can be replaced with statements that store displacements, block lengths and types.

However, the datatype constructors would require a number of arguments proportional to the number of values in the packing buffer. Furthermore, observe that the code to assemble these arguments would mirror the original packing code, effectively replacing packing code with code that packs displacements. Therefore, to improve efficiency and readability and to ensure we can exploit hardware features such as vector send, a sequence of optimization transformations are applied to yield *compact* datatypes.

We consider a datatype to be more compact than another datatype if it can be described with fewer arguments. A contiguous type is therefore more compact than a vector, which is more compact than an indexed type. That is, a contiguous type only requires two arguments (count and subtype), while a vector requires four (count, block length, stride and subtype). Furthermore, an indexed type requires $2n + 2$ arguments, where n is the number of blocks it describes. Compact datatypes result in higher performing code, as shown in section 6, and they also tend to be more readable.

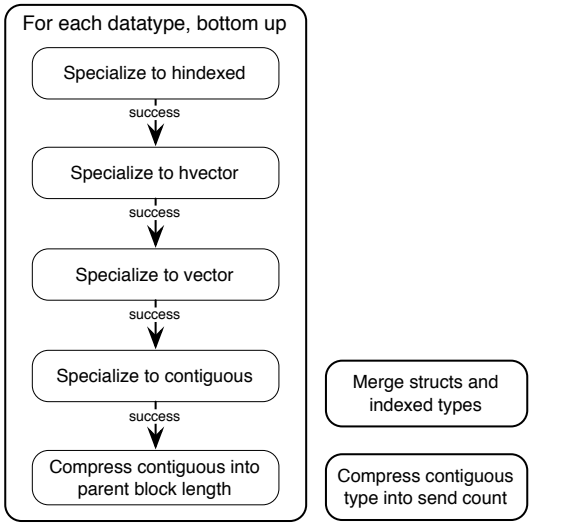


Figure 5. Figure showing the datatype optimization stages. Each datatypes is specialized, bottom up.

There are two types of optimizations: *specializing substitutions* that replace a more general datatype with a more specialized and compact one, and *datatype compressions* that merge datatypes.

Figure 5 shows the order in which the optimizations are applied. Each optimization makes the datatype strictly more compact. The sequence of specialization optimizations, as well as the "compress into parent block length" are applied to one datatype at a time, bottom up. The "compress into parent block length" optimization is performed interleaved with the specialization optimizations, because it exposes additional specialization opportunities. If any optimization, except from a specialization to hindexed, fails then the algorithm continues to the next datatype.

Once every datatype has been processed, the "Merge structs and indexed types" and "Compress contiguous type into send count" optimizations are applied to further reduce the datatypes.

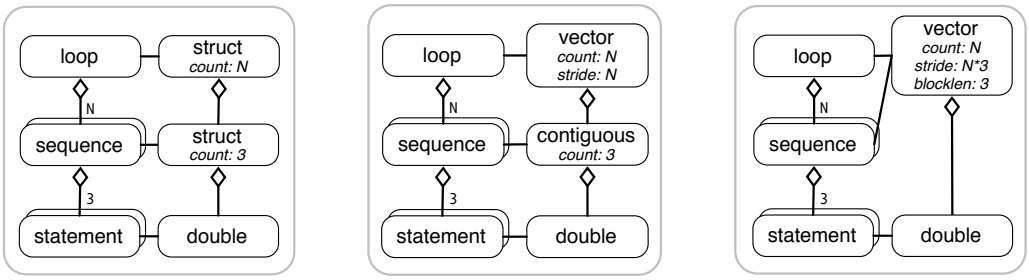
Figure 4(b) shows the datatypes in our running example after the specialization optimizations have been applied. The figure is slightly modified for presentation, as parent block length compression is interleaved with the specialization optimizations.

Figure 4(c) shows the final IR after the compression optimizations have also been applied. The only compression that was applicable in this case was the compress contiguous types parent into block length compression. This compression replaced the contiguous type with a corresponding increase in vector block length. Note that since the stride of vector datatypes is given in multiples of the subtype, the stride bas increased by a factor equal to the count of the contiguous type (i.e., from `N` to `N*3`).

The following sections each describe one of the seven datatype optimizations that are depicted in figure 5. The specialization substitution optimizations are described in sections 4.2.1–4.2.4, while the datatype compression optimizations are described in sections 4.2.5, 4.2.6 and 4.2.7.

4.2.1 Specialize to hindexed

After IR construction every datatype is a struct. The first specialization therefore transforms struct datatypes to hindexed datatypes.



(a) Initial IR (b) After specialization (c) After compression

Figure 4. Datatype IR after each transformation stage for the left Border Exchange example from figure 2.

Hindexed datatypes are similar to structs as both specify a byte displacement for each block. However, they add the constraint that all subtypes must be the same datatype. While struct datatypes allow a different subtype to be specified for each block, hindexed types only allow a single subtype to be specified for all the blocks. This transformation therefore requires that every struct block has the same subtype. If they do then the struct is specialized to an hindexed datatype. Note that vector or contiguous types that represent packing groups inside a loop may or may not represent a set of the same type, depending on whether the values that construct them change between loop iterations.

In our running example the struct representing the packing sequence can be specialized to an hindexed type as every subtype is an `MPI_DOUBLE`. The struct representing the packing loop can also be specialized to an hindexed type as all of its subtypes are the same.

4.2.2 Specialize to hvector

Hvector datatypes describe data layouts as a sequence of blocks with a fixed byte stride between each block. This is a significant constraint over hindexed datatypes, where each block is described by independent byte displacement. However, it also leads to a significant reduction in the number of arguments required to specify the datatype (down to four from linear), and hence a significant improvement in its efficiency. Three preconditions must be met before hindexed datatypes can be specialized to hvector types:

Every packing statement must access the same source array, struct or scalar variable C, like most imperative languages, does not provide any guarantees for the relative locations of distinct variables in memory. Since hvectors require the same stride between the source data structure locations of each pair of consecutive packing groups, these locations must be in the same array, struct or scalar variable.

All block lengths must be the same size Hvector datatypes only allow one block length to be specified, which applies to every block. The block length of each block must therefore be the same, and this can be established by symbolically comparing the block length expressions of each hindexed block.

The packing group must not contain a conditional If the packing group contains a conditional statement then it can not be specialized to an hvector type.

There must be a fixed distance between the locations in the source data structure accessed by every pair of consecutive packing statements As mentioned above, hvectors describe data blocks that are located a fixed stride apart. The distance between the source data structure locations of each consecutive packing group described by the hvector must therefore be fixed. There are two cases to consider when determining whether this is the case, namely packing sequences and packing loops.

Packing Sequences For a packing sequence to be specialized to an hvector, the distance between each consecutive pair of packing groups must be the same. As described in section 4.1.3 the IR construction phase computes these distances. The hindexed to hvector specialization pass can therefore simply compare the distance between every consecutive pair of packing groups symbolically.

Consider the packing sequence from our running example in figure 2(b), which consists of three packing statements. The location index expression of the first statement is $i \cdot N \cdot 3$, the index of the second statement $i \cdot N \cdot 3 + 1$, and the index expression of the third statement $i \cdot N \cdot 3 + 2$. The distances between consecutive packing statements are therefore:

$$((i \cdot N \cdot 3 + 1) - (i \cdot N \cdot 3), (i \cdot N \cdot 3 + 2) - (i \cdot N \cdot 3 + 1))$$

Determining that the expressions describing two distances are the same requires expression simplification, which is key to the effectiveness of our algorithm. However, expression simplification is widely used in mathematical software, and some techniques have been discussed in the literature [3]. One simple method is to convert each expression into a polynomial and then put the polynomial into a canonical form. This can be done efficiently, and allows comparison in linear time by comparing every term independently. After symbolic simplification the above expression becomes $(1, 1)$, and since the distances are the same, the packing statements can be represented by an hvector. The hvector stride is the expression describing the distances, multiplied with the size of the elements in source data structure. In the above example the stride therefore becomes `sizeof(double)`.

Packing Loops As defined earlier, packing loops only contain one sub-group. However, the nature of loops means this sub-group may be executed many times. Loop induction variables are variables whose value changes from one loop iteration to the next. If the location displacement expression of the loop's sub-group is an index expression that contains an induction variable, then it will describe different memory locations in each iteration.

Linear induction variables are a sub-class of induction variables whose values increase or decrease by a fixed amount in each loop iteration. That is, their progression follows an arithmetic sequence. A linear induction expression is an expression that only contains loop constant expressions and linear induction variables, and the linear induction variables are only combined through addition and subtraction. The values of such an expression also follow an arithmetic sequence. If the location index expression of the loop's sub-group is a linear induction expression, then it can be represented by an hvector. The stride of the arithmetic sequence described by the linear induction expression becomes the hvector's stride.

Non-linear induction variables are induction variables whose progression follows some non-arithmetic sequence. One example is a variable that is multiplied by itself in each loop iteration, thus

following a geometric sequence. Another example is a variable that is assigned a different value from an array in each iteration of a loop (i.e., the array is indexed using one or more induction variables). If non-linear induction variables are used in the location index expression of a loop's sub-group, then that loop can not be described by an hvector.

Linear induction variables can be summarized using standard approaches [5, 15], and a simple adaptation of these approaches can be used to summarize induction expressions. These algorithms summarize every linear induction variable as an affine function of the form $f(i) = c_1 biv + c_2$, where biv is a *basic induction variable*, and c_1 and c_2 are both loop constant expressions. c_1 describes the induction variables stride relative to the stride of the basic induction variable, and c_2 describes the induction variables value in the first iteration of the loop.

A basic induction variable is a variable that is incremented or decremented by a constant expression in each loop iteration. The variable `i` in figure 2(b) is an example of a basic induction variable, and is described with the affine function $1 \cdot i + 0$, i.e., in terms of itself. Note that c_2 is 0 as the induction variable has the value 0 in the first iteration of the loop. This can be determined using a standard reaching definitions analysis [15]. Variables that are not described in terms of themselves are called dependent induction variables. An example of a dependent induction variable would be `j = i * 2`. The affine function of `j` would be $2i + 0$.

Given a list of the affine expressions describing linear induction variables, we compute their *stride*. The stride of a basic induction variable is simply the constant expression c_1, while the stride of a dependent induction variable is the stride of c_1 multiplied with the stride of its basic induction variable. Induction expressions are summarized the same way as dependent induction variables, and their strides are computed the same way.

The algorithm calculates the stride for the location of every packing statement contained by the packing loop, or any of its packing subgroups. If these strides are constant between loop iterations, and the stride of every packing subgroup is the equal, then the packing loop's hindexed datatype can be specialized to an hvector.

In the example in figure 2(b) the loop is a packing loop that contains a packing sequence. The location of the packing sequence is the same as the location of its first packing statement, namely $(grid, i \cdot N \cdot 3)$. As mentioned above, `i` is a basic induction variable. The affine function describing the stride of the location's index expression is therefore $N \cdot 3 \cdot i + 0$. Since the stride of `i` is 1, the stride of the location's stride expression is $N \cdot 3$.

Since the packing sequence location is a linear induction expression the packing loop can be described by a hvector datatype, with a stride of `N*3*sizeof(double)`.

4.2.3 Specialize to vector

Vectors are like hvectors, with the exception that their strides are given in number of elements (multiples of the subtype extent) instead of bytes. That is, a vector's stride can be multiplied by its subtype's extent to yield its stride in bytes.

This places an additional constraint on the packing code. For an hvector to be specialized to a vector, the hvector's stride must be divisible by the extent of it's subtype. One test for this is to evaluate whether each term of the hvector's stride contains a one or more factors that are equal to the subtype's extent.

If the hvector stride is divisible by the subtype extent, then the hvector can be specialized to a vector and the vector's stride is the quotient of the division. A datatype's extent defines the distance from its first element to its last.

In our running example the stride of the hvector describing the packing sequence is `sizeof(double)`. Furthermore, the extent of the MPI_DOUBLE subtype is also `sizeof(double)`. The hvector stride is clearly divisible by the subtype extent, and the hvector can therefore be specialized to a vector with a stride of `1`.

The running example also contains a packing loop that has so far been specialized to an hvector. The hvector's subtype is the datatype describing the packing sequence, and its extent is $3 \cdot sizeof(double)$. This can clearly divide the stride of the packing loop's hvector, which has a stride of $N \cdot 3 \cdot sizeof(double)$, and the quotient of the division is N. The packing loop's hvector can therefore be specialized to a vector with stride `N`.

4.2.4 Specialize to contiguous

The last IR transformation along the main specialization chain is to specialize vectors to contiguous datatypes. A contiguous datatype describes elements that are laid out contiguously in memory. The precondition for this transformation is therefore that the stride of the vector, which describes the distance between the start of each block, is equal to its block length.

In our example, the stride and block length of the vector describing the packing sequence are both 1. It can therefore be specialized to a contiguous type. However, the vector describing the packing loop has a stride of N and a block length of 1, and since these differ, no further specialization is possible.

Figure 4(b) shows the Datatype IR after the specializations described in section 4.2.1-4.2.4 has been applied. The packing loop is described by a vector with count `N`, block length `1`, and stride `N`. Its subtype is the packing sequence, which is described by a contiguous datatype with count `3` and whose subtype is MPI_DOUBLE.

4.2.5 Compress contiguous into parent block length

Contiguous types describe elements laid out contiguously in memory. However, struct datatypes provide a block length argument that specifies the number of contiguous elements in each block.

If a struct has a contiguous subtype, then that subtype can be folded into the struct's block length argument. This involves three operations. First, the subtype of the struct is replaced with the subtype of the contiguous type. Second, the block length argument of the struct is symbolically multiplied by the count of the contiguous type. Third, the contiguous type is deleted.

This transformation is applied to a struct datatype and its subtype after the subtype has been fully specialized, but *before* the parent struct is specialized. It is therefore unnecessary to fold contiguous types into the block length of other datatypes, as folding will have taken place before specialization.

4.2.6 Merge structs and hindexed types

If a struct or hindexed datatype has a subtype of the same type, then the subtype can be merged with the parent datatype. This involves making the subtype's subtype the subtype of the parent, symbolically multiplying the subtype's count with the parent's count, and deleting the subtype. This transformation is applied after all datatypes have been fully specialized. For example, consider the case where the IR in figure 4(a) could not be specialized at all. In that case the merge struct compression would merge the two structs, resulting in one struct of size $N * 3$.

4.2.7 Compress contiguous types into send count

Send functions provide a count argument to specify the number of layouts described by the datatype to send. If the top datatype is a contiguous type then it can be folded into the send count. This involves multiplying the send count by the count of the contiguous type, replacing the send call's datatype with the contiguous type's subtype, and deleting the contiguous type. This compression transformation is applied after all specialization has been completed.

```
MPI_Datatype hidx_t;
MPI_Aint displacements[N]; int blocklen[N];
unsigned int idx = 0;
MPI_Aint first_addr;
MPI_Get_address(&grid[0][0][0], &first_addr);
for(int i=0; i < N; i++) {
  MPI_Get_address(&grid[i][0][0], &displacements[idx]);
  displacements[idx] -= first_addr;
  blocklen[idx] = 3;
  idx++;
}
MPI_Type_create_hindexed(idx, blocklen, displacements,
    MPI_DOUBLE, &hidx_t);
```

Figure 6. An hindexed type equivalent to the vector in figure 2(c).

4.2.8 Other Optimizations

Figure 5 shows the datatype specialization chain leading to the best possible datatype — a contiguous type. However, most datatypes we have seen in real-world code are not contiguous leading to one of steps in the chain not passing its preconditions.

In these cases alternative specialization is sometimes possible. For instance, struct or an hindexed types can sometimes be specialized to a struct or an indexed type with a fixed block length. The precondition for this step is equivalent to one of the precondition for going from hindexed to hvector, since hvectors also have fixed block lengths. Furthermore, byte strided hindexed types can sometimes be specialized to element strided indexed types. This specialization is similar to the specialization from hvector to vector and its precondition is equivalent.

MPI provides support for indexed and blocked indexed datatypes. However, since these do not seem to be prevalent in the code we studied, we do not discuss them in detail in this presentation.

5. Packing Code Replacement

Once the algorithm from the previous section has generated a datatype that represents the packed data, three steps remain to replace the packing code with the datatype. These steps are described in the following sections, in the context of C and MPI. The steps would be similar for other environments.

5.1 Datatype Emit

First the datatype must be emitted. Figure 2 shows how the packing code with a strided access pattern at lines 2–6 in 2(b), is replaced with the code that constructs a vector datatype at lines 1–3 in 2(c). Datatype declaration, vector constructor call and commit statements are inserted before the call to `MPI_Send` and a free statement after. If nested datatypes are needed to replace the packing code, then these are emitted one after another, such that each datatype is declared and constructed before datatypes that use it.

This scheme is sufficient for hvector, vector and contiguous datatypes, but struct and hindexed types require more care. These have a number of arguments proportional to the number of packing groups they describe, and must therefore be produced with logic that is equivalent to those groups. For example, an hindexed datatype representing the packing loop in figure 2(b), requires an hindex construction loop with the same structure. However, instead of packing values from the grid into the packing buffer, the hindexed construction code computes and packs byte displacements into a displacement buffer. Figure 6 shows the resulting code, assuming the three packing statements are still specialized to a contiguous type and compressed into the block lengths.

Consider a case where the three packing statements are only specialized to a vector. In this case, the vector construction code could have been hoisted out of the loop. Such hoisting is important for the performance of the datatype construction code, since it prevents the same datatype from being reconstructed in each loop iteration. The requirement for hoisting datatype construction code out of a loop is that no loop induction variables are used to construct it. This can easily be established, given that these variables are already known from the IR construction.

```
static MPI_Datatype vec_t;
static int init = 0; static int _N;
if (N != _N || !init) {
  if (init) MPI_Type_free(&vec_t);
  init = 1; _N = N;
  MPI_Type_vector(N, 3, N * 3, MPI_DOUBLE, &vec_t);
  MPI_Type_commit(&vec_t);
}
```

Figure 7. Boilerplate code to prevent redundant type regeneration

An issue with the datatype construction codes presented thus far is that they regenerate constant datatypes for every send. In most cases we have observed, packing codes pack the same elements every time they are executed. Datatype construction is not without cost, especially for hindexed and struct types, and to be performing the code should not unnecessarily regenerate datatypes. One solution, that is easy to automate, is to insert lazy initialization code that only constructs a datatype the first time it is executed, and when the arguments used to construct it are changed. Thus, the datatype, an init flag, and the previous values of each variable used to construct the datatype must be cached. In C, these can be put in the static code segment, by declaring them using the `static` keyword. Figure 7 demonstrates this approach for the construction of the vector in figure 2(c).

5.2 Packing Code Consumer Rewrite

Once the datatype has been emitted a refactoring tool should attempt to alter the code that consumes the packed data to instead use the datatype. In the compiler scenario the compiler must do this, or else there is no use in emitting the datatype in the first place.

This step requires matching the packing code to a packing code consumer, and then ensuring the packing buffer is not overridden in between. Furthermore, for unpacking codes in MPI, the number of received elements must match the size of the unpacked data. However, if it does not then a tool can append an overflow buffer to the datatype to catch redundant received data.

Figure 2 shows how the `MPI_Send()` is rewritten to use the datatype. The address argument is set to the address of the first packing statement in the first iteration of the loop, the count argument to 1, and the datatype argument to the new datatype, `vec_t`.

5.3 Dead Packing Code Elimination

After the datatype has been emitted and the packing code consumers have been altered, dead packing code remains which should be removed through a dead packing code elimination pass. Dead code elimination has been extensively studied before [15], and is therefore not covered in this text.

6. Evaluation

To evaluate the usefulness our algorithm we answer the following experimental questions in this section:

1. Is the algorithm applicable to real-world codes?
2. Does the algorithm produce compact datatypes?
3. Is the algorithm fast enough for interactive use?

6.1 Methodology

To answer the first question, we implemented our algorithm in a refactoring tool for C with MPI. We then applied the tool to every packing and unpacking code block in a version of the MG, LU, BT and SP applications in the NAS parallel benchmarks. Since these applications are written in Fortran, we first ported their communication kernels to C. Every send and receive in these applications sends packed data, or receives data that is then unpacked.

To answer the second question, we first inspected the datatypes and confirmed that they were compact. We then collected statistics from the NAS parallel benchmarks, showing how many datatypes each optimization applied to. These results also provides insight into the structure of the data communication in these applications.

To evaluate the efficiency of datatypes on contemporary systems, and to evaluate the importance of optimizing datatypes, we ran the benchmarks with optimized and unoptimized datatypes produced by our tool on the Cray XT-5 partition of the Jaguar system at Oak Ridge National Lab. We then compared the execution time with that of the original packing code.

To answer the third question, we measured the time it took for our refactoring tool to transform packing and unpacking codes.

6.1.1 Implementation

The refactoring tool was implemented as a prototype refactoring plugin on top of the Eclipse CDT IDE.[1] We used Eclipse 3.6.1 and a patched version of CDT 7.0.2. The tool has two modes.

In the first mode the user marks packing code and a send call, or unpacking code and a receive call, in the text editor. She then asks Eclipse for transformation suggestions through the Eclipse Quick Assist system. Eclipse then invokes our plugin, and if the code passes our preconditions then one of the options will be REPLACE PACKING CODE WITH DATATYPE CODE. If the user selects this option then our refactoring plugin analyses the packing code and produces a matching datatype representation. It then rewrites the packing code to equivalent code that constructs and uses a datatype, as was demonstrated in figure 2.

The second mode works mostly the same way as the first mode, but instead of selecting packing code and a send call, the user only selects packing code. As the tool can not distinguish between packing and unpacking code when a send or a receive call is not marked, it provides two options. The first option converts packing code, while the second option converts unpacking code. Depending on which option the user selects, the tool generates a datatype describing the packed or unpacked data. It also creates a comment with a send or a receive call, where the three first arguments (address, count and datatype) are provided. It is then up to the user to copy the datatype to the site of the send or receive, and to copy the arguments in the comment into the send or receive call. This mode was useful when porting the NAS applications, where the applications often start receives and then performed sends, before calling `MPI_Wait()` and unpacking the received data.

The tool is implemented in Java and consist of nearly 7000 SLOC. It operates on the CDT AST and performs expression simplification using the Symja Open Source Library version 0.0.13 [21].

6.1.2 NAS

The NAS parallel benchmarks are a set of programs designed to evaluate the performance of parallel supercomputers [1]. The source code for seven of these are made available with NAS 3.2. Of these MG, LU, SP and BT contains point-to-point communication (sends and receives) of mostly non-contiguous data, where packing code is used to serialize the data instead of datatypes [13]. We applied our tool to the packing code of every send call, and the unpacking code of every recv call, in these applications.

The MG application uses a multigrid method to compute the solution of the three-dimensional scalar Poisson equation. LU is a simulated CFD application which uses symmetric successive over-relaxation (SSOR) to solve a block lower triangular-block upper triangular system of equations resulting from an unfactored implicit finite-difference discretization of the Navier-Stokes equations in three dimensions. SP an BT each solves three sets of uncoupled systems of equations, using multi-partition schemes [1].

These applications contain a total of 48 sends and 48 receives. Every send sends packed data, and the data from every receive is unpacked. Thus, NAS contains a total of 96 packing and unpacking codes. The packed/unpacked data structures are 2–6 dimensional arrays, while the packing buffers are arrays with 1 to 3 dimensions. The packing code range from simple loops containing one packing statement, to several deeply nested loops containing multiple packing statements, and packing from multiple data structures.

Appendix A provides a case study of how the tool converts one of the packing codes in the LU benchmark to datatypes.

6.2 Results

The NAS applications contain 96 packing code blocks. Half of these pack data that is sent using `MPI_Send` or `MPI_Isend`, while the other half unpack data received through `MPI_Recv` or `MPI_Irecv`. We applied our tool to every packing and unpacking code block.

In 90 cases our tool was able to transform the code with little or no programmer assistance, yielding a success ratio of 94%. In 30 cases precondition two was not met, but in each case it was established through one loop split. In all of the unpacking codes, except four in LU, the data was received using `MPI_Irecv` and the unpacking code was located after the corresponding `MPI_Wait`. In these cases we used our tool in its second mode, saving us from constructing the datatype and only leaving us the work of copying the datatype to the site of the receive call. This highlights an important strength of a refactoring tool. Unlike a compiler, where each transformation must be fully automated or not at all, a refactoring tool supports a workflow where automated and manual transformations are *interleaved*. Thus, a refactoring tool can automate transformations that are hard for a programmer (create datatypes), while the programmer can perform tasks that are easy for her (move code).

In one application, MG, 12 sends and 12 *unpacking* codes were mapped to a single `MPI_Recv`, through the use of `MPI_ANY_SOURCE`. In this case additional effort was required to duplicate the `MPI_Irecv`. Furthermore, in some cases more data was sent to the `MPI_Irecv` than was unpacked by the packing code. This required the programmer to extend the datatypes produced by our tool with a contiguous buffer to hold the redundant received data. However, this feature can easily be implemented as an extension to our tool if this corner case shows up in other codes.

Our algorithm was not applicable in six cases, which were all unpacking codes in the SP application. The reason why these codes could not be represented by datatypes was that the "unpacking buffers" were never unpacked, but instead used in the computation.

Figure 8 present statistics for the 90 packing and unpacking codes that our tool transformed. As illustrated by the pie chart in the center, most of the packing groups are structured, with only 25% requiring an hindexed or struct type. The hindexed and struct types were mostly located at the top of the datatype hierarchies. As shown in the left pie chart, 41% of the resulting datatypes were of these types. All contiguous types were compressed into blocklen or send/recv count arguments and are therefore not explicit in the emitted code. The bar chart shows that the MG and LU communication is very structured, and mostly transfers vector and contigu-

[1] The refactoring tool to convert packing code to datatypes, as well as the experimental data can be found at `http://web.mit.edu/fredrikk/www/datatypes.html`

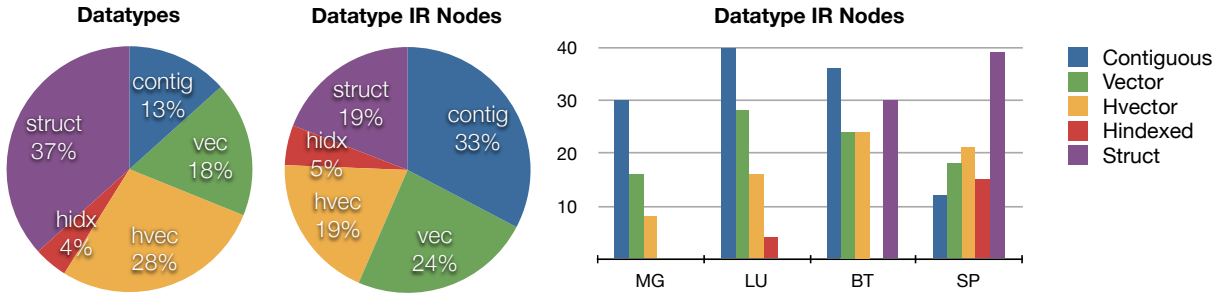


Figure 8. Tool statistics when applied to 90 packing and unpacking codes in the NAS parallel benchmarks. The pie chart on the left shows the type distribution of the resulting 90 top-level datatypes that are used in communication calls. The pie chart in the center shows the distribution of the 361 datatype IR nodes that contribute to these hierarchical datatypes. Each IR node represents one packing group. The bar chart on the right shows the number of each IR node type for each of the benchmark applications.

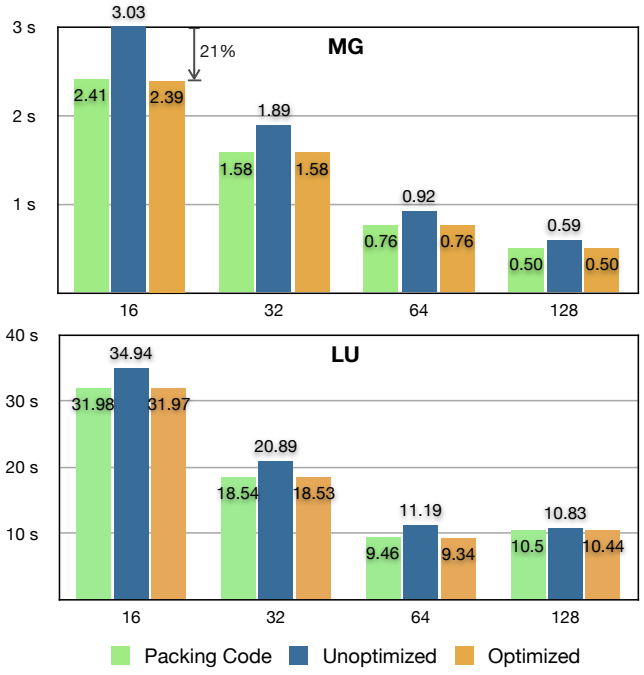


Figure 9. Application running times on 16, 32, 64 and 128 processors of Jaguar. Running times were collected for the original packing code, for unoptimized datatypes (structs) produced by our tool, and for optimized datatypes (figure 8) produced by our tool.

ous types. SP is the least structured with a high number of struct and hindexed types. It was also the only benchmark that contained packing codes that did not meet our algorithm's preconditions.

The unoptimized Java implementation of our algorithm took on average 35 ms to transform each packing/unpacking code block, with the fastest transformation taking 5 ms and the slowest taking 112 ms. Thus, the refactoring tool appears instantaneous to the user. The measurements were performed on a MacBook Pro 6.2, with an Intel Core i7 processor clocked at 2.66 GHz. Transformation times were correlated with the number of packing groups that had to be specialized, and the complexity of their expressions. We manually inspected the packing code produced by our tool and confirmed that it was correct and compact. Furthermore, we ran the resulting code on two clusters and used the verification feature of the NAS applications to verify that the tool output was correct.

Figure 9 shows the running time for three versions of the MG and LU applications, running on 16, 32, 64 and 128 processors of the Jaguar system. The first version is the original MG and LU code which contains packing code. In the other versions our tool has been used to replace packing code with datatypes. In the first of these we configured our tool to only apply the merge structs optimization, so that each packing and unpacking code was replaced with one struct datatype. Thus, our tool inserted logic to pack the addresses of each element to be sent into an displacement array as demonstrated in figure 2(c). In the final version our tool applied all optimizations, producing the datatypes from figure 8.

As shown, the unoptimized datatypes perform significantly worse than the packing code on Jaguar, causing the running time of the applications to increase by up to 26%. However, once the full range of optimizations described in sections 4.2.1–4.2.7 has been applied, the performance of the applications matches or outperforms the packing code versions by up to 1.28%. Phrased another way, our datatype optimizations increase the performance of these applications with up to 21% over non-optimized datatypes. These results shows that the datatype optimizations presented in this paper are essential for automatically generated datatypes. Note that the reported running time is for the whole applications. In these applications communication only take up a small part of the running time, so the pure communication speedups are higher.

Keep in mind that Jaguar is a contemporary system, that does not utilize hardware support to speed up non-contiguous transfers. As discussed in section 2, several researchers have shown that with proper support the performance of non-contiguous transfers can be improved by up to 6.8X. We provide these experiments to show that datatypes can match the performance of packing code on contemporary systems. However, the real value of datatypes will be on future machines, and this work will help programmers be ready.

7. Future Work

Numerous lines of research could be inspired by this work. We will first discuss future work that are natural extensions of this work. We will then suggest research projects that are related to datatypes, but that are not direct extensions of this work.

This work presents a general algorithm, that would fit as well into a compiler as a refactoring tool. A natural next step is therefore to implement it in a compiler, so that programming models without a means of specifying datatypes can take advantage of advance HW transfer capabilities. A compiler can not rely on the user, and must automatically match packing statements to sends and unpacking statements to receives. The latter is complicated by asynchronous receives, where the packing code is located after the wait barrier. In such cases a compiler must match receives to

their corresponding wait. Lastly, a compiler implementation should perform enabling transformations, such as loop split and inlining, and post-transformations such as dead packing code elimination.

The author's experience and anecdotal evidence suggests that datatypes are hard to construct. This motivated an automated approach, even for an environment such as MPI that expose datatypes to the programmer. However, a user study to quantify the benefits of automation has not yet been conducted, and would provide a useful contribution to the datatype literature.

Our algorithm can handle packing code that packs data that is transferred through point-to-point communication calls, moved through RDMA operations, and written to files using IO operations. Future work includes handling code that packs data that is transferred using collectives such as MPI's AllToAll. These AllToAll collectives take a datatype, and sends one of these data layouts to each rank that participate in the exchange. The AllToAll contains an implicit contiguous type specifying the relative location of the data going to each rank. However, in certain cases, such as matrix transpose, it is necessary to send overlapping non-contiguous data to the different participants. Since the current version of MPI's AllToAll implicitly exchanges contiguous layouts, the only way to support this that is known to the authors is to tweak the extent of the top datatype [10]. An extension to our algorithm would detect this case and convert the top hvector type resulting from this use into a contiguous type by tweaking the extent of its subtype. Knowing that a contiguous type is implicit in the AllToAll, it could then be removed before code is emitted. Future work also includes investigating whether similar extensions are needed for other collectives.

Our algorithm targets packing code and converts it to datatype code. Packing code is the most common way to transfer non-contiguous data if datatypes are not used or available. The reason for this is that packing code reduces the number of messages to one, thus minimizing latency. However, an alternative approach to transfer non-contiguous data is to send one message per non-contiguous block, perhaps using asynchronous pipelined messages. Future work could extend our algorithm to detect this pattern in user code, and convert the non-contiguous sends to a single send that uses datatypes to describe the layout. This would up free the runtime to use pipelined messages where this makes sense, to pack data where this is more efficient (e.g., on a system without support for asynchronous sends), or non-contiguous transfer support in hardware where this is available.

The work presented in this paper converts packing or, by extension, unpacking code to datatype code by only considering one side of the communication. This imposes a strict requirement on the ordering of the elements in the layout described by the datatype. That is, the datatype must describe a layout where every element are sent or received in the same order as they appeared in the original packing buffer. However, if sends and receives could be matched then both sides of the communication could be changed simultaneously. This would free the algorithm from the ordering constraint and turn it into an optimization process. Datatypes element ordering could then be optimized both for our notion of datatype efficiency (least number of arguments), and for improved memory access patterns.

Datatypes are hard to construct or change, and the programmer has little help if the first attempt was incorrect. There are no effective way to debug datatypes, and the programmer is left guessing what might have gone wrong. A useful future work is therefore to build a datatype debugger. Such a debugger could print the memory locations that are sent. Moreover, with sufficient knowledge of the user data structure, it could even show the layout of the data described by the datatype textually or graphically. Information about the user data structure could either be automatically extracted from the program, if it is regular enough, or provided by the user. Consider code that sends the south face of a three dimensional mesh. A datatype debugger could present a 3D visualization showing the mesh with the face in question colored red.

Many current systems with MPI support do not provide hardware facilities for non-contiguous transfers. Furthermore, systems that do provide such capabilities often do not exploit them. This creates a vicious circle: Programmers do no use datatypes as they do not provide performance gains; and vendors do not optimize datatypes as they are not used by current codes. Such systems instead pack the data inside the MPI library by interpreting the datatype. Such an interpreted solution often performs slightly worse than user provided C packing code that has been compiled to machine code. There are two promising ways to get around this. The first is to provide an MPI-aware offline compiler that can compile datatypes to optimized machine packing code. The second approach is to embed an online datatype compiler into the `MPI_Type_Commit` function call. This approach is preferable as it is transparent to the programmer, easily deployable, and can provide increased performance without recompiling application code. The online compiler would be able to produce code that matches compiled user packing code, or that even outperforms it if it can take advantage of runtime information to produce more optimal machine code than is possible at compile time. Furthermore, the cost of compilation over the lifetime of an execution is unlikely to be a bottleneck, as datatypes are typically generated once and then re-used in every communication.

8. Conclusions

We have described in this paper an algorithm for transforming packing or unpacking messages into a compact datatype that describes the data layout. We have also described and evaluated an implementation of this algorithm within the Eclipse framework, as a refactoring tool for parallel codes using MPI. We expect that such refactorings will facilitate the correct use of compact MPI datatypes thus enabling programmers to reduce communication overheads. We plan to make this refactoring available within Eclipse, as part of a refactoring environment for parallel code tuning.

Transfers of noncontiguous data are prevalent in many parallel codes; efficient "scatter-gather" capabilities that optimize such transfers and reduce the amount of memory traffic generated by such transfers will be essential in future HPC architectures. We expect that future high performance network interfaces and/or future memory controllers will have enhanced scatter-gather capabilities, enabling direct transfer of noncontiguous data from memory to the network interface or to the CPU. Datatypes provide a mean of communication between the executing code and such a smart scatter-gather engine. The transformations we outlined will facilitate this communication — as part of a compiler or a run-time capability.

References

[1] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, 1995.

[2] D. Bonachea. Gasnet specification, v1.1. Technical Report UCB/CSD-02-1207, U.C.Berkeley, 2002.

[3] B. Buchberger and R. Loos. Algebraic simplification. *Computer Algebra - Symbolic and Algebraic Computation*, pages 11–44, 1982.

[4] S. Byna, W. Gropp, X.-H. Sun, and R. Thakur. Improving the performance of MPI derived datatypes by optimizing memory-access cost. In *International Conference on Cluster Computing*, 2003.

[5] J. Cocke and K. Kennedy. An algorithm for reduction of operator strength. *Communications of the ACM*, 20(11):850–856, 1977.

[6] K. D. Cooper, L. T. Simpson, and C. A. Vick. Operator strength reduction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(5):603–625, September 2001.

[7] A. Danalis, K.-Y. Kim, L. Pollock, and M. Swany. Transformations to parallel codes for communication-computation overlap. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2005.

[8] D. Goujon, M. Michel, J. Peeters, and J. Devaney. AutoMap and AutoLink: Tools for communicating complex and dynamic data-structures using MPI. In *Lecture Notes in Computer Science*, volume 1362, 1997.

[9] W. Gropp, E. Lusk, and D. Swider. Improving the performance of MPI derived datatypes. In *MPI Developer's Conference*, 1999.

[10] T. Hoefler and S. Gottlieb. Parallel zero-copy algorithms for fast fourier transform and conjugate gradient using MPI datatypes. In *Recent Advances in the Message Passing Interface*, 2010.

[11] InfiniBand Trade Assoc. Infiniband architecture specification 1.1.

[12] F. Kjolstad and M. Snir. Ghost cell pattern. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, 2010.

[13] Q. Lu, J. Wu, D. Panda, and P. Sadayappan. Applying MPI derived datatypes to the nas benchmarks: A case study. In *Proceedings of the International Conference on Parallel Processing Workshops*, 2004.

[14] J. Moses. Algebraic simplification a guide for the perplexed. In *Proceedings of the second ACM symposium on Symbolic and algebraic manipulation*, 1971.

[15] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[16] J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. volume 1586 of *Lecture Notes in Computer Science*.

[17] R. W. Numrich and J. Reid. Co-Array Fortran for parallel programming. *ACM SIGPLAN Fortran Forum*, 1998.

[18] OFED. `http://www.openfabrics.org/index.php`.

[19] R. Ross, N. Miller, and W. Gropp. Implementing fast and reusable datatype processing. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2003.

[20] G. Santhanaraman, J. Wu, and D. K. Panda. Zero-copy MPI derived datatype communication over InfiniBand. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2004.

[21] Symja, June 2011. URL `http://code.google.com/p/symja/`.

[22] N. Tanabe and H. Nakajo. Acceleration for MPI derived datatypes using an enhancer of memory and network. In *IEEE International Symposium on Parallel Distributed Processing Workshops*, 2010.

[23] W. Tansey and E. Tilevich. Efficient automated marshaling of C++ data structures for MPI applications. In *International Symposium on Parallel and Distributed Processing*, 2008.

[24] UPC Consortium. UPC language specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab, 2005.

[25] J. Worringen, A. Gaer, and F. Reker. Exploiting transparent remote memory access for non-contiguous- and one-sided-communication. In *Workshop on Communication Architecture for Clusters*, 2002.

[26] J. Wu, P. Wyckoff, and D. Panda. High performance implementation of MPI derived datatype communication over infiniband. In *IEEE International Symposium on Parallel and Distributed Processing*, 2004.

A. Datatype Extraction Case Studies

This section provides two case studies of how our tool analyzes and converts packing code to datatypes. These case studies are meant to supplement the rest of this report, and to help implementers. Both of the case studies are taken from the NAS parallel benchmarks [1], and highlights different aspects of our technique.

The first case study is presented in section A.1, and is taken from the NAS LU application. It demonstrates how a very regular border exchange code is converted to the Datatype IR and optimized. It then shows how the packing code is replaced with code to initialize the corresponding datatype, and also demonstrates how lazy initialization code can be added to ensure the datatype is not regenerated for every send operation.

The second case study is more complex and is taken from the SP application, which has the most irregular packing code of the applications in the NAS parallel benchmarks. The SP packing code contains an if statement, thus requiring an indexed type and also have datatypes whose construction depends on outer loop induction variables, thus preventing full hoisting. Section A.2 contains this case study. In the interest of brevity we have not shown the lazy initialization code for this case study, but it is similar to the one demonstrates for the first case study.

Note that we have chosen these case studies because they demonstrate different features of our tool, and they are selected to be significantly more complex than the illustrative example. The first demonstrate packing sequences, specialization, compression and lazy code insertion, while the second demonstrates code that can only be specialized to struct and hindexed types, as well as hoisting. The fact that both require loop splits is coincidental, and we would remind the reader that in the NAS benchmarks, only one third of the packing code blocks required loop splits.

A.1 LU Border Exchange

This section provides a case study of how our tool analyzes and converts a packing code block from the NAS LU application to datatype code.

The `exchange_3.f` communication kernel in the NAS LU application contains four point-to-point send/receive pairs that perform border exchanges in the north-south and east-west directions. The borders are 2-dimensional faces of a 3-dimensional structured mesh. Each cell in the mesh has five values, so the resulting array is 4-dimensional.

Figure 10 shows the original Fortran code for the south border send. The loops pack two 2-dimensional faces into the packing buffer. The first face is packed into the `buf` packing buffer starting at location `1`, while the second face is packed into the same buffer starting at location `ny*nz`. The first face is packed by the statements on line 60–64, while the second face is packed by the statements on line 65–69.

Figure 11 shows the code ported to C. The original Fortran code used Fortran's convenient array syntax to pass a 4-dimensional array into the communication kernel. However, when this array is passed to C code it becomes a 1-dimensional pointer. In the ported code this pointer is accessed using the standard approach for treating 1-dimensional arrays as multi-dimensional arrays, which is to multiply each index with the inner dimensions. In practice a programmer might hide this indexing logic behind a macro or a function, in which case our tool would need to either expand the macro/inline the function, or analyze the macro/function to figure out the index expression. However, macro expansion and function inlining are known techniques and are not covered here. Furthermore, since our tool is a refactoring tool we can rely on the programmer to perform these operations.

As each iteration of the loops packs into two different parts of the packing buffer, this code does not satisfy the third precondition from section 4.1.1. To establish this precondition the programmer must therefore perform one loop split to produce the code in figure 12.

Once the third precondition has been established our tool can apply our algorithm to the packing code to construct an equivalent datatype. Our tool first summarizes the packing groups in the code, shown in figure 13(a). Once the packing groups have been constructed the tool assigns a struct datatype to every packing loop and packing sequence. These are then specialized and compressed as described in section 4.2, yielding the datatypes in 13(b). The inner packing sequences are described with contiguous datatypes that are compressed into the block length of the parent datatype (vec_t). The inner loop is specialized to a vector, while the outer loop can only be specialized to an hvector. Since both loops describe the same layout they can be described with the same datatypes (vec_t and hvec1_t). Finally, the hvector hvec_t, with a count of 2, describes the packing sequence containing the two loops.

The datatypes in the IR are used by our tool to emit the code in figure 14. Note that the tool does not currently perform dead-code elimination of the packing loops (lines 51–70 in figure 12), but for presentation purposes we have manually removed these. For the same reasons, we also added newlines between the construction of each datatype, although the tool does not do this for technical reasons.

One issue with the code in figure 14 is that it constructs a new datatype every time the south border is sent. This is costly and unnecessary as the datatype can often be reused. Since our tool is a refactoring tool we can leave to the user the task of ensuring the datatypes are not unnecessarily re-created. Such tasks are typically easy for users. However, a compiler that implements our algorithm can not rely on the user to modify its output, and should therefore address this problem. We therefore present two solutions to this problem. Both solutions employ lazy initialization to prevent unnecessary re-construction of datatypes.

Figure 15 shows the first solution, which we call reduced lazy initialization. The `hvec_t` datatype is added to the static code segment through the use of the `static` keyword. Furthermore, a static decision variable is used to only construct the datatype the first time the code is executed. Note that this solution requires that the variables that are used to construct the datatype, `ny`, `nz`, `isiz1` and `isiz2`, are not changed before any subsequent execution of the communication code. Using this solution the user, or an automated tool that insert this check, would have to verify that this is the case. This is usually easy for a user, who knows a lot about the usage patterns of her variables. However, it would be hard for a tool to determine this, as it would require whole program analysis.

The second solution requires no additional analysis. We call this the full lazy initialization strategy, and it is the one shown in figure 7 and discussed in section . This is possible by adding additional code to dynamically check at runtime that none of these variables have been changed between executions of the communication code. Figure 16 shows these checks for the south border send example. The value of the variables that are used to construct the datatype are stored in static variables when the datatype is constructed. If the communication code is executed again with different values for these variables, then the datatype is re-constructed. Adding this scaffolding is trivial for a tool, and we plan to add this feature as an option in our refactoring tool.

```
56            do k = 1,nz
57              do j = 1,ny
58                ipos1 = (k-1)*ny + j
59                ipos2 = ipos1 + ny*nz
60                buf(1,ipos1) = g(1,nx-1,j,k)
61                buf(2,ipos1) = g(2,nx-1,j,k)
62                buf(3,ipos1) = g(3,nx-1,j,k)
63                buf(4,ipos1) = g(4,nx-1,j,k)
64                buf(5,ipos1) = g(5,nx-1,j,k)
65                buf(1,ipos2) = g(1,nx,j,k)
66                buf(2,ipos2) = g(2,nx,j,k)
67                buf(3,ipos2) = g(3,nx,j,k)
68                buf(4,ipos2) = g(4,nx,j,k)
69                buf(5,ipos2) = g(5,nx,j,k)
70              end do
71            end do
72
73            call MPI_SEND( buf,
74           >               10*ny*nz,
75           >               dp_type,
76           >               south,
77           >               from_n,
78           >               MPI_COMM_WORLD,
79           >               IERROR )
```

Figure 10. LU South Border Exchange (Fortran)

```
54 for (k = 1; k <= nz; k++) {
55   for (j = 1; j <= ny; j++) {
56     ipos1 = (k-1)*ny + j;
57     ipos2 = ipos1 + ny*nz;
58     buf[(ipos1-1)*5    ] = g[(k-1)*(isiz2+4)*(isiz1+4)*5 + (j+1)*(isiz1+4)*5 + (nx)*5      ];
59     buf[(ipos1-1)*5 + 1] = g[(k-1)*(isiz2+4)*(isiz1+4)*5 + (j+1)*(isiz1+4)*5 + (nx)*5   + 1];
60     buf[(ipos1-1)*5 + 2] = g[(k-1)*(isiz2+4)*(isiz1+4)*5 + (j+1)*(isiz1+4)*5 + (nx)*5   + 2];
61     buf[(ipos1-1)*5 + 3] = g[(k-1)*(isiz2+4)*(isiz1+4)*5 + (j+1)*(isiz1+4)*5 + (nx)*5   + 3];
62     buf[(ipos1-1)*5 + 4] = g[(k-1)*(isiz2+4)*(isiz1+4)*5 + (j+1)*(isiz1+4)*5 + (nx)*5   + 4];
63     buf[(ipos2-1)*5    ] = g[(k-1)*(isiz2+4)*(isiz1+4)*5 + (j+1)*(isiz1+4)*5 + (nx+1)*5    ];
64     buf[(ipos2-1)*5 + 1] = g[(k-1)*(isiz2+4)*(isiz1+4)*5 + (j+1)*(isiz1+4)*5 + (nx+1)*5 + 1];
65     buf[(ipos2-1)*5 + 2] = g[(k-1)*(isiz2+4)*(isiz1+4)*5 + (j+1)*(isiz1+4)*5 + (nx+1)*5 + 2];
66     buf[(ipos2-1)*5 + 3] = g[(k-1)*(isiz2+4)*(isiz1+4)*5 + (j+1)*(isiz1+4)*5 + (nx+1)*5 + 3];
67     buf[(ipos2-1)*5 + 4] = g[(k-1)*(isiz2+4)*(isiz1+4)*5 + (j+1)*(isiz1+4)*5 + (nx+1)*5 + 4];
68   }
69 }
70
71 MPI_Send( buf,
72     10*ny*nz,
73     MPI_DOUBLE,
74     south,
75     from_n,
76     MPI_COMM_WORLD);
```

Figure 11. LU Border Exchange after port to C

```
51 for (k = 1; k <= nz; k++) {
52   for (j = 1; j <= ny; j++) {
53     ipos1 = (k-1)*ny + j;
54     buf[(ipos1-1)*5    ] = g[(k-1)*(isiz2+4)*(isiz1+4)*5 + (j+1)*(isiz1+4)*5 + (nx)*5      ];
55     buf[(ipos1-1)*5 + 1] = g[(k-1)*(isiz2+4)*(isiz1+4)*5 + (j+1)*(isiz1+4)*5 + (nx)*5  + 1];
56     buf[(ipos1-1)*5 + 2] = g[(k-1)*(isiz2+4)*(isiz1+4)*5 + (j+1)*(isiz1+4)*5 + (nx)*5  + 2];
57     buf[(ipos1-1)*5 + 3] = g[(k-1)*(isiz2+4)*(isiz1+4)*5 + (j+1)*(isiz1+4)*5 + (nx)*5  + 3];
58     buf[(ipos1-1)*5 + 4] = g[(k-1)*(isiz2+4)*(isiz1+4)*5 + (j+1)*(isiz1+4)*5 + (nx)*5  + 4];
59   }
60 }
61 for (k = 1; k <= nz; k++) {
62   for (j = 1; j <= ny; j++) {
63     ipos2 = (k-1)*ny + j + ny*nz;
64     buf[(ipos2-1)*5    ] = g[(k-1)*(isiz2+4)*(isiz1+4)*5 + (j+1)*(isiz1+4)*5 + (nx+1)*5     ];
65     buf[(ipos2-1)*5 + 1] = g[(k-1)*(isiz2+4)*(isiz1+4)*5 + (j+1)*(isiz1+4)*5 + (nx+1)*5 + 1];
66     buf[(ipos2-1)*5 + 2] = g[(k-1)*(isiz2+4)*(isiz1+4)*5 + (j+1)*(isiz1+4)*5 + (nx+1)*5 + 2];
67     buf[(ipos2-1)*5 + 3] = g[(k-1)*(isiz2+4)*(isiz1+4)*5 + (j+1)*(isiz1+4)*5 + (nx+1)*5 + 3];
68     buf[(ipos2-1)*5 + 4] = g[(k-1)*(isiz2+4)*(isiz1+4)*5 + (j+1)*(isiz1+4)*5 + (nx+1)*5 + 4];
69   }
70 }
71
72 MPI_Send( buf,
73     10*ny*nz,
74     MPI_DOUBLE,
75     south,
76     from_n,
77     MPI_COMM_WORLD);
```

Figure 12. LU Border Exchange after loop split

```
Packing Sequence
  Packing Loop
    Packing Loop
      Packing Sequence
        Packing Statement
        Packing Statement
        Packing Statement
        Packing Statement
        Packing Statement
  Packing Loop
    Packing Loop
      Packing Sequence
        Packing Statement
        Packing Statement
        Packing Statement
        Packing Statement
        Packing Statement
```

(a) Packing Groups

```
hvector (hvec_t)
  hvector (hvec1_t)
    vector (vec_t)
      contiguous (compressed)
        MPI_DOUBLE
        MPI_DOUBLE
        MPI_DOUBLE
        MPI_DOUBLE
        MPI_DOUBLE
  hvector (hvec1_t)
    vector (vec_t)
      contiguous (compressed)
        MPI_DOUBLE
        MPI_DOUBLE
        MPI_DOUBLE
        MPI_DOUBLE
        MPI_DOUBLE
```

(b) Datatypes

Figure 13. LU Border Exchange IR

```
54 MPI_Datatype vec_t;
55 MPI_Type_vector(ny, 5, 5 * (4 + isiz1), MPI_DOUBLE, &vec_t);
56
57 MPI_Datatype hvec1_t;
58 MPI_Type_create_hvector(nz, 1, 5 * (4 + isiz1) * (4 + isiz2) * sizeof (double), vec_t, &hvec1_t);
59
60 MPI_Datatype hvec_t;
61 MPI_Type_create_hvector(2, 1, 5 * sizeof (double), hvec1_t, &hvec_t);
62 MPI_Type_commit(&hvec_t);
63
64 MPI_Send(&g[5 * nx + 10 * (4 + isiz1)], 1, hvec_t, south, from_n, MPI_COMM_WORLD);
65 MPI_Type_free(&hvec_t);
```

Figure 14. LU Border Exchange after tool conversion to datatypes, and user dead code elimination of the packing loops

```
54 static MPI_Datatype hvec_t;
55 static int init = 0;
56 if (!init) {
57   init = 1;
58
59   MPI_Datatype vec_t;
60   MPI_Type_vector(ny, 5, 5 * (4 + isiz1), MPI_DOUBLE, &vec_t);
61
62   MPI_Datatype hvec1_t;
63   MPI_Type_create_hvector(nz, 1, 5 * (4 + isiz1) * (4 + isiz2) * sizeof (double), vec_t, &hvec1_t);
64
65   MPI_Type_create_hvector(2, 1, 5 * sizeof (double), hvec1_t, &hvec_t);
66   MPI_Type_commit(&hvec_t);
67 }
68
69 MPI_Send(&g[5 * nx + 10 * (4 + isiz1)], 1, hvec_t, south, from_n, MPI_COMM_WORLD);
```

Figure 15. LU Border Exchange with the reduced lazy initialization strategy (assumes `ny`, `nz`, `isiz1` and `isiz2` are not redefined between executions)

```
54 static MPI_Datatype hvec_t;
55 static int init = 0, ny_v, isiz1_v, nz_v, isiz2_v;
56 if (ny != ny_v || isiz1 != isiz1_v || nz != nz_v || isiz2 != isiz2_v || !init) {
57   if (init) MPI_Type_free(&hvec_t);
58   init = 1;
59   ny_v = ny;
60   isiz1_v = isiz1;
61   nz_v = nz;
62   isiz2_v = isiz2;
63
64   MPI_Datatype vec_t;
65   MPI_Type_vector(ny, 5, 5 * (4 + isiz1), MPI_DOUBLE, &vec_t);
66
67   MPI_Datatype hvec1_t;
68   MPI_Type_create_hvector(nz, 1, 5 * (4 + isiz1) * (4 + isiz2) * sizeof (double), vec_t, &hvec1_t);
69
70   MPI_Type_create_hvector(2, 1, 5 * sizeof (double), hvec1_t, &hvec_t);
71   MPI_Type_commit(&hvec_t);
72 }
73
74 MPI_Send(&g[5 * nx + 10 * (4 + isiz1)], 1, hvec_t, south, from_n, MPI_COMM_WORLD);
```

Figure 16. LU Border Exchange with the full lazy initialization strategy

A.2 SP East Neighbor Transfer

This section provides a case study of how Our tool analyzes and converts one of the packing codes from the NAS SP application.

The file `copy_faces.f` contains the function `copy_faces`, which swaps the border values of the set of cells contained on this processor with each neighbor. Figure 17 shows excerpts from the original Fortran code. Line 70–90 contains the packing code. Note that the values to be sent to each of the neighbors are packed before any send is issued, and they share two outer loop, starting on line 70 and 71. Two loop splits (one for each outer loop) followed by code motion to the site of the `MPI_Isend()` per packing code block is therefore needed.

Figure 18 shows the packing code after these transformation has been applied to the C port. Lines 98–115 contains the packing code: 5 nested loops. The loops on line 102–104 contains array accesses in the loop bound expressions. These are treated as loop constants with respect to the loops they are used in, provided the arrays are never assigned to inside the respective loop and that the array subscripts does not contain variables that are induction variables of that loop. On line 101 an if statement is used to decide if the next three dimensional array slice should be packed. This if statement prevents the loops on lines 99 and 100 to be specialized to hvector types.

Figure 19 and 20 shows the result of applying our tool to this code. The three inner loops are converted to contiguous, vector and hvector types on lines 340–344 (the contiguous type has been compressed into the vector's block length). Note that these have been hoisted out of the previously outer loop on line 353, but not out of the loop on line 339. This is because the index variable of the loop on line 339, `c` is used to construct them.

The fourth loop contained an if statement, so it could not be specialized past hindexed. Note that it *could* be specialized to an hidexed type for the same reason that the contiguous, vector and hvector subtypes could be hoisted out of its construction loop on line 353. That is the construction of the subtypes does not depend on the induction variables in the loop on line 354 (`m`), and thus represent the same hvector for every hindex block.

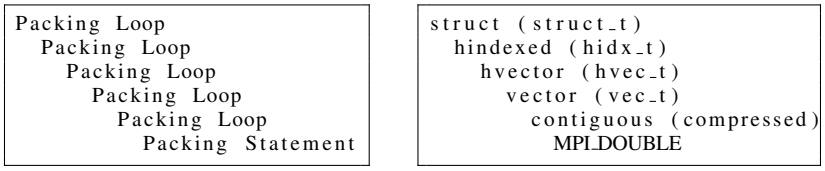
Finally, the outermost loop could not be specialized at all. This is because the hindexed type in every iteration of the loop starting on line 339 is different, as their type hierarchies are constructed using the index variable of the loop on line 339.

```
70          do  c = 1, ncells
71             do   m = 1, 5
72
73 c-----------------------------------------------------------------------
74 c              fill the buffer to be sent to eastern neighbors (i-dir)
75 c-----------------------------------------------------------------------
76                 if (cell_coord(1,c) .ne. ncells) then
77                    do   k = 0, cell_size(3,c)-1
78                       do   j = 0, cell_size(2,c)-1
79                          do   i = cell_size(1,c)-2, cell_size(1,c)-1
80                             out_buffer(ss(0)+p0) = u(i,j,k,m,c)
81                             p0 = p0 + 1
82                          end do
83                       end do
84                    end do
85                 endif
86
87                 ...
88
89 c-----------------------------------------------------------------------
90 c            m loop
91 c-----------------------------------------------------------------------
92              end do
93
94 c-----------------------------------------------------------------------
95 c          cell loop
96 c-----------------------------------------------------------------------
97          end do
98
99          ...
100
101         call mpi_isend(out_buffer(ss(0)), b_size(0),
102      >                 dp_type, successor(1),   EAST,
103      >                 comm_rhs, requests(6), error)
```

Figure 17. SP East Neighbor Transfer (Fortran)

```
98 p0 = 0;
99 for (c = 1; c <= ncells; c++) {
100   for (m = 1; m <= 5; m++) {
101     if (cell_coord[(c-1)*3] != ncells) {
102       for (k = 0; k <= cell_size[(c-1)*3 + 2]-1; k++) {
103         for (j = 0; j <= cell_size[(c-1)*3 + 1]-1; j++) {
104           for (i = cell_size[(c-1)*3]-2; i <= cell_size[(c-1)*3]-1; i++) {
105             out_buffer[ss[0]+p0-1] = u[(c-1)*5*(KMAX+4)*(JMAX+4)*(IMAX+4) +
106                 (m-1)*(KMAX+4)*(JMAX+4)*(IMAX+4) + (k+2)*(JMAX+4)*(IMAX+4) +
107                 (j+2)*(IMAX+4) + (i+2)];
108
109             p0 = p0 + 1;
110           }
111         }
112       }
113     }
114   }
115 }
116
117 MPI_Isend(&out_buffer[ss[0]-1], b_size[0],
118     dp_type, successor[1  -1], EAST,
119     comm_rhs, &requests[6]);
```

Figure 18. SP East Neighbor Transfer after port to C, loop split and code motion



(a) Packing Groups (b) Datatypes

Figure 19. SP East Neighbor Transfer IR

```
330 p0 = 0;
331
332 MPI_Datatype struct_t;
333 MPI_Aint displacements[ncells];
334 int blocklen[ncells];
335 MPI_Datatype types[ncells];
336 unsigned int idx = 0;
337 MPI_Aint first_addr;
338 MPI_Get_address(&u[8 + cell_size[0] + 2 * IMAX + 2 * (4 + IMAX) * (4 + JMAX)], &first_addr);
339 for(c = 1;c <= ncells;c++){
340   MPI_Datatype vec_t;
341   MPI_Type_vector(cell_size[-2 + 3 * c], 2, 4 + IMAX, MPI_DOUBLE, &vec_t);
342   MPI_Datatype hvec_t;
343   MPI_Type_create_hvector(cell_size[-1 + 3 * c], 1, (4 + IMAX) * (4 + JMAX) * sizeof (double),
344       vec_t, &hvec_t);
345
346   MPI_Datatype hidx_t;
347   MPI_Aint displacements1[5];
348   int blocklen1[5];
349   unsigned int idx1 = 0;
350   MPI_Aint first_addr1;
351   MPI_Get_address(&u[8 + cell_size[-3 + 3 * c] + 2 * IMAX + 2 * (4 + IMAX) * (4 + JMAX) +
352       5 * (-1 + c) * (4 + IMAX) * (4 + JMAX) * (4 + KMAX)], &first_addr1);
353   for(m = 1;m <= 5;m++){
354     if(cell_coord[(c - 1) * 3] != ncells){
355       MPI_Get_address(&u[8 + cell_size[-3 + 3 * c] + 2 * IMAX + 2 * (4 + IMAX) * (4 + JMAX) +
356           5 * (-1 + c) * (4 + IMAX) * (4 + JMAX) * (4 + KMAX) +
357           (-1 + m) * (4 + IMAX) * (4 + JMAX) * (4 + KMAX)], &displacements1[idx1]);
358       displacements1[idx1] -= first_addr1;
359       blocklen1[idx1] = 1;
360       idx1++;
361     }
362   }
363   MPI_Type_create_hindexed(idx1, blocklen1, displacements1, hvec_t, &hidx_t);
364
365   displacements[idx] = first_addr1 - first_addr;
366   blocklen[idx] = 1;
367   types[idx] = hidx_t;
368   idx++;
369 }
370 MPI_Type_create_struct(idx, blocklen, displacements, types, &struct_t);
371 MPI_Type_commit(&struct_t);
372
373 MPI_Isend(&u[8 + cell_size[0] + 2 * IMAX + 2 * (4 + IMAX) * (4 + JMAX)], 1,
374     struct_t, successor[1 - 1], EAST, comm_rhs, &requests[6]);
375 MPI_Type_free(&struct_t);
```

Figure 20. SP East Neighbor Transfer after tool conversion to datatypes, and user dead code elimination of the packing loops